Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research
Computer Science and Engineering

Report Number: WUCSE-2002-16

2002-07-09

Session Configuration and Network Design in Programmable Networks

Sumi Choi

This proposal addresses two problems in programmable networks. Specifically, we are interested in networks that can dynamically deploy applications and session-specific plugins within network routers, to provide advanced commination services. In the first half of the proposal, we present a general approach to the problem of configuring application sessions that require intermediate processing in programmable networks. In the second half, we discuss how to provision a programmable network for such session by placing and dimensioning link bandwidth and processing resources in an efficient way.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Choi, Sumi, "Session Configuration and Network Design in Programmable Networks" Report Number: WUCSE-2002-16 (2002). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1134

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Washington University in St. Louis

School of Engineering & Applied Science

Session Configuration and Network Design in Programmable Networks

Sumi Choi
syc1@arl.wustl.edu

WUCS-2002-16

July 9, 2002

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

This proposal addresses two problems in *programmable* networks. Specifically, we are interested in networks that can dynamically deploy applications and session-specific plugins within network routers, to provide advanced communication services.
In the first half of the proposal, we present a general approach to the problem of configuring application sessions that require intermediate processing in programmable networks. In the second half, we discuss how to provision a programmable network for such sessions by placing and dimensioning link bandwidth and processing resources in an efficient way.

1. Introduction

Computer networks have evolved to provide convenience in diverse areas of everyday life. Applications developed in such networks, particularly in the Internet, have various purposes and objectives. In order to support the dynamics and the variety of such applications, research in *active networks* [22][2] has been exploring the programmability of networks. In programmable networks, applications can be designed so that some of their processing occurs not only on end systems but also on routers. This feature opens up a set of flexible ways to develop and deploy new services. For example, consider an application that requires specific encryption and decryption to be applied to its session data transmitted between two remote corporate networks. On behalf of the application session, the processing component on routers can be programmed to perform the encryption and the decryption. Now, advances in technology are making it possible to build such networks by enabling general purpose processing in network routers. Many research efforts have been undertaken to develop programmable routers [7] and methods for operating programmable networks.

In this work, we study two problems that are tightly related to managing resources in programmable networks. The first problem deals with mapping application sessions onto appropriate network resources. For the example given above, the session is mapped onto links for the data flow and the routers where the encryption and decryption are performed on the data. The requirements, which include processing cycles, memory, or the ability to execute a particular program, are used to identify the set of routers capable of the processing. Given such a characterization, we model the problem in a graph space and study how to optimally configure individual sessions. In Section 3, we provide a formal definition and details of the problem.

Second, we study the network configuration problem in programmable networks. While the first problem deals with dynamically configuring application sessions given a set of available resources, the second part focuses on how to deploy resources to satisfy expected needs for resources. We focus on situations where the network topology is fixed and the anticipated traffic of the target applications are given, and then attempt to determine the capacity required for each resource. We also study to best place processing resources, when number of processing nodes is limited. The formal description and the definition of the problem is given in Section 4.

2. Related Works

Several approaches have been proposed to realize the idea of programmable networks. One such approach is called *active networking* [2],[22]. Active networking is best known with a variant called *capsule model*, where a packet may include a specific program and is intercepted by a router that executes the program on the packet data before forwarding it to the next hop router. With respect to coding the intermediate programs, several programming languages have been proposed as in [11], [25].

More practical variants of active networking involve pre-installation of programs at

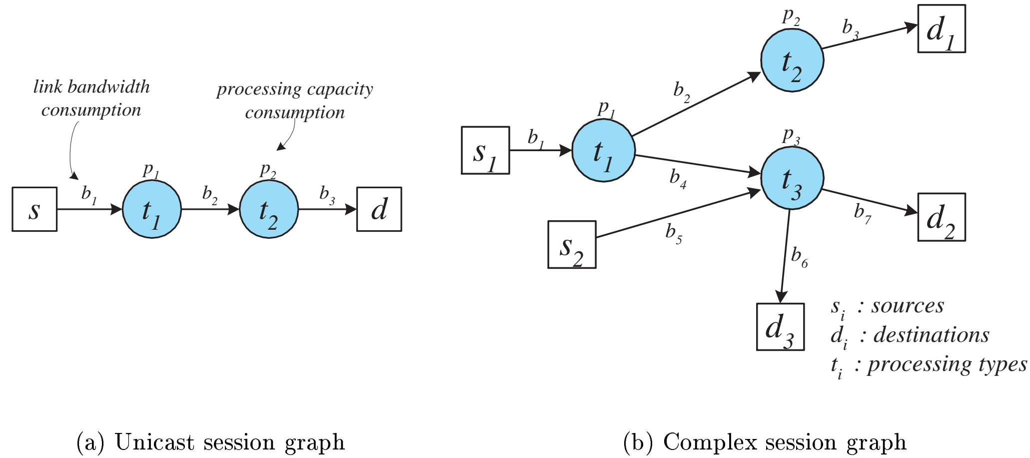
routers in advance of packet arrivals instead of carrying programs in packets. The installation procedure can be done with a signalling protocol that loads a trusted program into routers, and the program can be applied to subsequent packets [7]. Programs can also be installed by network administrators at designated routers in the network as a part of supporting new services. After installing the programs, the applications providing the services can have their packets routed through the routers that are installed with corresponding programs. In this case, the installation of such programs must be done as a part of the service deployment, which plays a crucial role in the performance of the applications and the utilization of the network. Chae et al. proposed a distributed mechanism called the Iterative Gather-Compute-Scatter(IGCS) computation model [3] that enables applications to access various network status and attributes, and furthermore identify the nodes or links where target services need be installed or deployed. Another method providing distributed deployment of services, which extends the IGCS method, is proposed in [10], which particularly focus on an hierarchical network model that aggregates the network information in order to support multiple autonomous networks. We also discuss the problem of the service deployment in Section 4, where, however, the problem is considered in the network designing phase given expected resource usages for the target services.

For both variants of active networking, routers need a software for executing the programs that are either conveyed by or installed for application data. Such software infrastructure has been exploited in many research efforts [1], [18], [23], [24].

Programmable networks expand the concept of resource in networks by allowing applications to use processing resources at routers in addition to links through which data is transmitted. *Darwin* project [4] is focused on the management of this broader set of network resources. One of the core mechanisms in Darwin system is a resource or service broker called *Xena* which discovers and selects the resources that are necessary and (near) optimal for service requests from applications. Once the resources are identified, a signaling mechanism called *Beagle* conveys signaling messages to reserve the resources for the corresponding application. Darwin also contains a mechanism that manages and adapts the resource usages at each resource, which is based on a Java code module called a *delegate*.

Among the mechanisms in Darwin, Xena implements the algorithm to determine the resources to be configured for each service request of an application or an application session. While Xena expresses the algorithm as a 0-1 integer programming problem to solve more general forms of application sessions, our session configuration described in Section 3 provides a more efficient algorithm for the most popular subsets of session forms and may be combined with Xena for better performance.

Another component necessary to realize programmable networks is the application programming interface(API) through which individual applications request and utilize resources. A successful example of such interfaces is the BSD socket interface that has been widely used in UNIX operating systems for applications communicating through Internet. As programmable networks expand the concept of network resources to include the processing capabilities of routers, we need a new API that supports the new resource type. Active pipe [13] proposes an API for programmable networks, that abstracts heterogenous application requirements and interacts with lower level entities that manage network resources.



(a) Unicast session graph

(b) Complex session graph

Figure 1: Session graphs

3. Configuring Sessions in Programmable Networks

This section details the problem of mapping an application session onto resources in the network, where intermediate processing is required. For example, consider a video transmission application, whose goal is to have the video data sent from a sending device and transformed at an intermediate router so that an incompatible receiving device can view it. The configuration of this session should identify the set of links used to form the data path and the router which will do the video transformation.

In general, the task of configuring a session is composed of identifying the session format, particularly the communication and the processing that occur in the session, and then mapping the format onto the resources in the network that would fulfill the goals of the session.

Depending on the purposes and the goals, network applications can take different formats. In this work, the session formats are described with the terminals sending and/or receiving data, the data flows among the terminals, and the processing applied to the data flows. We use a graph model to express each session format, where the nodes stand for the terminals or the processing steps, and the edges stand for the data flows. Each edge in the graph is associated with the link bandwidth consumed by the corresponding data flow. Similarly, each non-terminal node is associated with the processing capacity consumed by the corresponding processing step. Figure 1(a) shows an example that describes a unicast session with two steps of intermediate processing. In the figure, the terminal nodes are s and d, and the processing steps are $t1$ and $t2$, while the edges specify the data flows from one terminal through the other terminal. The bandwidths consumed by the data flows are b_1, b_2 and b_3, and the processing capacities are p_1 and p_2. We refer to the graph describing a session format as the session graph. We have another example of a session graph in Figure 1(b) that involves more terminals, processing and data flows. In general, a session graph forms a directed graph that involves one or more terminal nodes. While arbitrary topologies are possible, we focus our attention here on paths and trees.

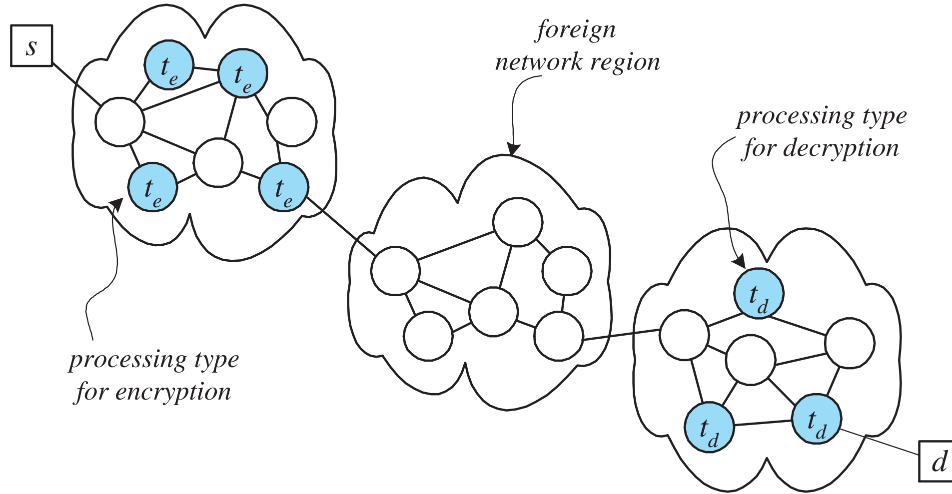


Figure 2: Processing types with location constraints

We also describe the network as a graph. While the session graph describes only the properties of a session, the network graph describes the entire network, which is composed of terminals, routers, and physical links that connect them. For programmable networks, some routers are specially labeled as processing nodes representing programmable routers. We may further categorize the processing nodes so that each of them is labeled with the types of processing that it can handle. The categorization is partially determined by the system specifications on the routers, in which properties like processing power, memory size or software capabilities may be considered. Processing can also be categorized by application constraints. For instance, recall the example of the encryption and decryption application. In this case, the application intends to perform the encryption before the data leaves the network region that includes the sender and to perform the decryption after the data enters the network region that includes the receiver. To reflect this intent, we label all processing nodes in the sender's network region to express the type of the nodes where the encryption can be done. Similarly, we label all processing nodes in the receiver's network region for decryption. Figure 3 shows the two regions and the associated processing types.

Figure 3 shows an example of a network graph, where terminals are drawn as squares and router nodes are drawn as circles. Each processing node is also labeled with the processing types (t_1, t_2, t_3) that it can handle.

Now that we have described the sessions and the network as graphs, the task of configuring a session is a matter of mapping the session graph onto the network graph, where the terminals are mapped to the same terminals, the processing steps to processing nodes of the appropriate types, and finally the connections between two nodes that are adjacent in the session graph to paths that connect the corresponding nodes in the network graph. Following this principle, we can consider configuring the session graphs in Figure 1 onto the network graph in Figure 3. The configurations are given in Figure 4.

Meanwhile, in the networks where resources like processing nodes and links are costly and limited commodities that are shared by multiple parties, sessions incur expenses when

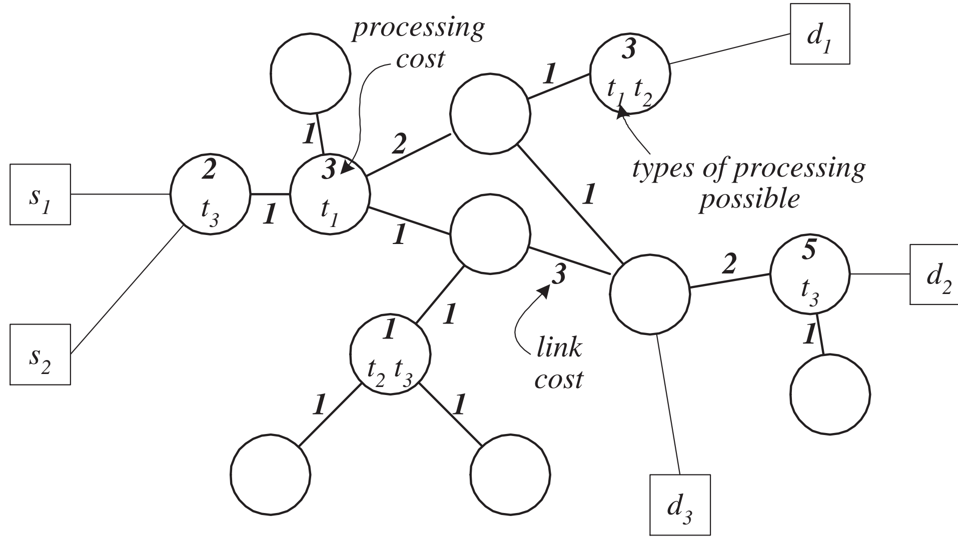


Figure 3: Network graph

they consume resources that are configured and assigned to them. In this work, we refer to the expense pertaining to consuming a unit capacity at a resource as its *unit cost*. While the actual measure for the cost may vary depending on the network models, we limit it to be a positive value as illustrated by the numbers labeling processing nodes and edges in Figure 3.

Starting from the following section, we formally describe the problem of configuring sessions with the optimal cost and present efficient solutions for the most important specific case of the problem (sessions which define paths). Later, we also discuss heuristics targeting other cases, and issues related to resource capacities.

3.1. Configuration for Generic Sessions

In this section, we define the configuration cost of sessions and give a formal statement for the optimal session configuration problem based on the cost. Previously, we introduced the cost associated with each resource as it is configured for a session. Given a mapping that identifies the resources to be configured for a session, we can compute the cost at each resource as the product of its unit cost and the capacity of the resource consumed by the session. The configuration cost of the session is then defined as the sum of the costs of all resources designated by the mapping. Now, we attempt to find the mapping that results in the least cost configuration, which is the goal of the optimal session configuration problem.

In PROBLEM 3.1, we formally state the optimal session configuration problem given a network graph $G = (V, E)$ and a session graph $G_s = (V_s, E_s)$.

PROBLEM 3.1 *Session Configuration Problem*
Given: *A directed session graph* $G_s = (V_s, E_s)$*, with a type* $t(u)$ *and a capacity requirement* $p(u)$ *for each vertex* $u \in V_s$*, and a bandwidth requirement* $b(u, v)$ *for each edge* $(u, v) \in E_s$*. Also, a directed network graph* $G = (V, E)$ *with a type set* $T(u)$ *for each node* $u \in V$*.*

(a) Unicast session configuration

(b) Complex session configuration

Figure 4: Session configuration

In addition, the unit costs, $c(u)$ and $c(u, v)$, are given for each vertex and edge as a positive value.
Find: *A location function $l : V_s \rightarrow V$ and a routing function $r : E_s \rightarrow 2^E$ that satisfy*

$$\forall u \in V_s, \quad t(u) \in T(l(u)) \tag{1}$$

$$\forall (u, v) \in E_s, \quad r(u, v) \text{ is a simple path in } G \text{ from } l(u) \text{ to } l(v) \tag{2}$$

and that minimize the cost $C_{l,r}$ where

$$C_{l,r} = \sum_{u \in V_s} c(l(u))p(u) + \sum_{(u,v) \in E_s} c(r(u, v))b(u, v) \tag{3}$$

In the problem, the capacity consumption of the target session is expressed as requirements for the configured resources in order to guarantee that the capacities to be consumed are actually available on the resources. For the moment, we assume that the network resources have unlimited capacities and focus on optimizing the configuration cost.

The mapping that designates the resources for a session is defined by the location function l and the routing function r, where the function l maps each node in G_s to a node in G, and the function r maps each edge in G_s to a path in G. The functions should also satisfy the conditions, (1) $\sim$ (2), where Condition (1) ensures that selected nodes can perform the required processing and Condition (2) ensures that adjacent nodes in the session graph are connected by a path in the network graph. It also limits the path to be simple in order to avoid unnecessary link usage.

Finally, we define the configuration cost $C_{l,r}$ to be the total sum of the costs of all resources used by the session. Therefore, the optimal mapping must minimize the configuration cost $C_{l,r}$.

Among graph problems, the *graph embedding problem* [17] is most closely related to the session configuration problem. In the graph embedding problem, an embedding of a

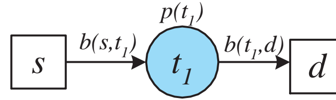


Figure 5: Unicast with single step processing

graph G onto another graph G' is defined as a mapping from the former to the latter, where each distinct node in G is mapped to a distinct node in G', and each edge in G to an edge in G'. The graph embedding problem has been used for designing parallel algorithms on interconnection networks, which are composed of processors in distributed memory machines. Here, the algorithms are represented by graphs where each node stands for a processing step, and each edge stands for the sequence and/or the data flow between steps. Then, the embeddings are used to implement and operate the algorithms in the network.

Often in more realistic situations, the edge mapping in the graph embedding problem is relaxed so that an edge in G is allowed to be mapped to a *path* in G'. This formulation is called weak graph embedding and closely resembles our session configuration problem. In fact, the weak graph embedding problem is a special case of the session configuration problem, where the session graph and the network graph have a single processing type. In addition, the optimal weak embedding problem is defined in the same way as the optimal session configuration problem with values assigned to the edges and the nodes used in the embeddings.

Unfortunately, the optimal weak embedding problem is known to be $\mathcal{NP}$-hard [17] and thus, so is the general session configuration problem because it contains the optimal weak embedding problem.

Nevertheless, there are categories of session patterns, to which most application sessions belong, in which the optimal session configuration can be solved or closely approximated efficiently. We will discuss these cases next.

3.2. Configuring unicast sessions

Unicast is the most common session form. A unicast session takes the form of a path with a single source and destination plus zero or more intermediate processing steps. The session graph of a unicast session is shown in Figure 1(a) with two steps of intermediate processing.

When no processing is involved, the optimal configuration is identical to the least cost path between the terminals. For this particular case of unicast sessions, standard shortest path algorithms can be used to find the best session configuration.

Now, when intermediate processing is required, the configuration must also select processing nodes. As described in Problem 3.1, the function l maps the processing steps in the session graph to nodes with the corresponding processing types. Here, the configuration still forms a path from the source to the destination, where, in addition, the processing nodes designated by the function l are included as intermediate nodes. Figure 4(a) shows a configuration for the unicast session graph given in Figure 1(a), where the configured path

from source s to destination d passes through two designated processing nodes with the matching types.

The optimal configuration in this case should also have the least configuration cost, which now includes the cost of each of the configured processing nodes, in addition to the path cost. Because of the node selection, standard shortest path algorithms are not directly applicable to the problem. Nevertheless, the shortest path information is still crucial to solving the problem. Below, we illustrate a method that computes the least cost configuration for unicast sessions and discuss related issues. Initially, we focus on unicast sessions with single step processing, and then expand the discussion to multiple steps.

Let us assume a unicast session in the network graph $G = (V, E)$ with the source s, the destination d and one processing step of type t_1 to be done on the data flow from s to d, where R is the set of nodes, which can handle the processing. The session graph of this session is shown in Figure 5 with its bandwidth requirements and processing capacity requirement.

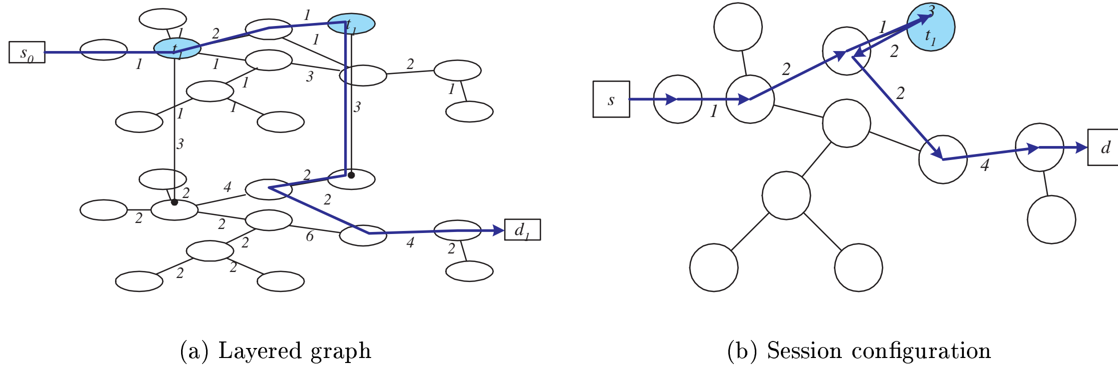
First, for each processing node r in R, compute the shortest paths from s to r and from r to d, and construct the configuration using the shortest paths as the data paths and the node r as the processing node. Also, compute the cost of this configuration by summing the cost of the shortest paths and the cost of the processing node r. Here, the cost of each link in the path from s to r is scaled up by the bandwidth consumption $b(s, t_1)$ from its unit cost, and similarly each link in the path from r to d is scaled up by $b(t_1, d)$. The cost of r is scaled up by $p(t_1)$. Note that this configuration gives the least cost given r as the processing node. Then, among all nodes in R, select the node that results in the smallest configuration cost when used as the processing node, and choose the associated configuration as constructed above.

In this method, the shortest paths can be found by computing the shortest path tree from the source s and another converging to the destination d. Using Dijkstra's algorithm, the time complexity is $O(|V| \log(|V|) + |E| + |R|)$.

However, there are issues with generalizing this method to handle more processing steps. Consider another unicast session, which now requires two processing steps. Let us denote the sets of nodes for the processing steps as R_1 and R_2. By applying the same method, we need to account for each pair of processing nodes in $R_1 \times R_2$ to configure the session. Now, $|R_1|+2$ shortest path trees need to be computed to obtain all the shortest paths required for the configurations. Then, $|R_1| \times |R_2|$ comparisons are needed to find the least cost configuration. If there are k steps, we need to compute and compare the cost associated with each choice for the processing nodes, where the number of choices is $O(|R_1| \times |R_2| \times ... \times |R_k|)$. In the worse case, the comparison takes $\Omega(n^k)$ time.

3.2.1. Layered networks for single step processing. In this section, we introduce a better alternative for solving the unicast session configuration problem. This method takes the problem given in the network graph into a different space, where the problem is solved as a conventional shortest path problem. Then, the result is brought back into the original network graph to obtain the final solution.

We illustrate the method, focusing on the transformation that converts the network graph into a new graph space called a "layered network". Let us reconsider the unicast



(a) Layered graph (b) Session configuration

Figure 6: Session configuration with a layered graph

session with a single processing step, where R is the set of processing nodes that are capable of the processing given the network graph $G = (V, E)$. For this session, the new method, which we will call the layered graph method, transforms the original network graph into a "two layer" graph.

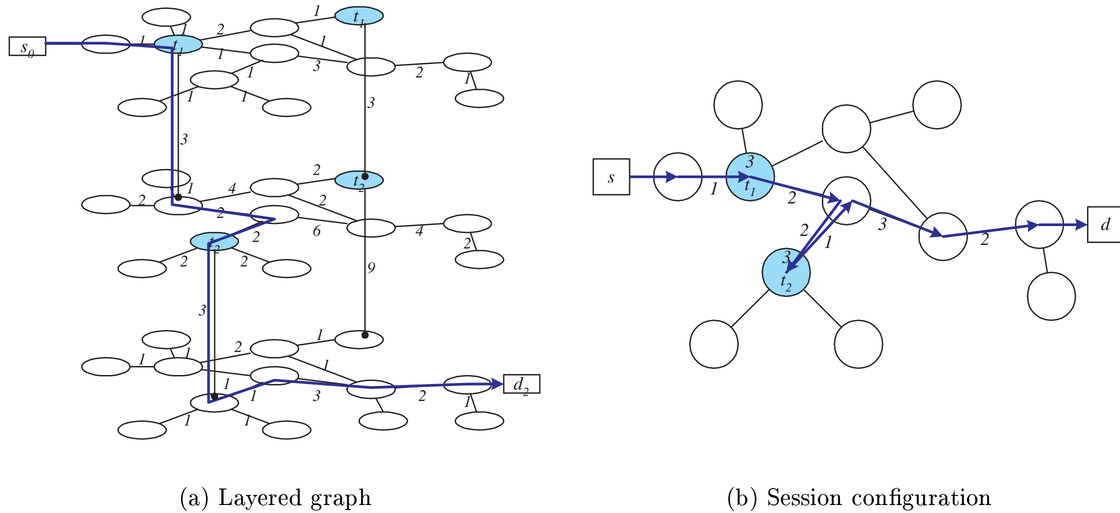
The layered network G' includes two copies of the network graph G. We refer to one copy as layer 0 and the other copy as layer 1. Also, for each node v in G, we denote the copy of the node in layer 0 as v_0 and the copy in layer 1, v_1. The cost of each edge in layer 0, say (u_0, v_0), is set to the product of the unit cost of the original edge (u, v) in the network graph and the bandwidth requirement $b(s, t_1)$, i.e. $c(u, v) \times b(s, t_1)$. Similarly, the cost of each edge (u_1, v_1) is set to $c(u, v) \times b(t_1, d)$.

Then, we complete the layered network G' by adding an inter-layer edge (r_0, r_1) for each processing node r in R. Here, the cost of (r_0, r_1) is set to the product of the unit cost of the node r and the processing requirement $p(t_1)$, i.e. $c(r) \times p(t_1)$. Figure 6(a) shows the layered graph transformed from the original graph in Figure 3 for sessions with processing type t_1, $b(s, t_1) = 1, b(t_1, d) = 2$, and $p(t_1) = 1$.

Given the new graph G', the layered graph method computes the least cost path from the node s_0, the copy of the source in layer 0, to the node d_1, the copy of the destination in layer 1. Note that G' only carries edge costs, so shortest path algorithms can be applied directly. Figure 6(a) also shows the least cost path in the layered graph.

For the final solution to the unicast session configuration problem, the least cost path in G' is mapped back to the network graph G as follows. For each regular edge involved in the path, we "project" it to the original edge in G. Similarly for each inter-layer edge in the path, we "project" it to the original processing node in G and mark it as the designated node for the processing requested in the session graph. The projection yields a legitimate configuration connecting the two terminals and containing the processing node on the path. The projected configuration of the least cost path in Figure 6(a) is given in Figure 6(b) with its configuration cost. We claim that this configuration gives the least cost and therefore is the optimal solution.

To prove our claim, let us assume that there is another configuration in the network graph with a smaller cost. We can map this configuration back to a path in the layered



(a) Layered graph

(b) Session configuration

Figure 7: Session configuration with a layered graph

graph by simply reversing the "projection" procedure. Now, this path has the same end points as the least cost path while having the same cost as the configuration from which it is mapped. Thus, it has a smaller cost than the least cost path, which is a contradiction. Hence, our claim holds.

The layered graph method has three parts, first constructing the layered network, second computing the least cost path in the layered network, and last projecting the path back to the original network. The first part can be implemented with $O(|V| + |E| + |R|)$ by iterating on the set of nodes and edges. The second part can be implemented to run in $O(|V| \log |V| + |E| + |R|)$ using Dijkstra's shortest path algorithm. Lastly, the projection can be done in time linear in the size of the path, which is $O(|V'|)$. Here, the dominant part is the shortest path computation. The comparison method introduced earlier has the same asymptotic time complexity, and may slightly outperform the layered graph method in a real implementation. However, the layered graph method scales better for multiple steps, as we discuss in the next section.

3.2.2. Layered network for multiple processing steps. In this section, we generalize the layered graph method for an arbitrary number of processing steps. Consider a unicast session that involves two terminals s and d and processing with k consecutive steps of types $t_1, t_2, ..., t_k$, where the bandwidth requirements are $b(s, t_1), b(t_1, t_2), ..., b(t_k, d)$, and the processing capacity requirements are $p(t_1), p(t_2), ..., p(t_k)$. For each step i, let us denote the set of processing nodes capable of the step as R_i.

We build the layered network in a similar way as for single step processing. For k steps, we make $k + 1$ copies of the original network where the copies are denoted layer 0 through layer k. For each link (u, v), we apply the scaled cost $c(u, v) \times b(t_i, t_{i+1})$ to its copy in layer i, where $t_0 = s$ and $t_{k+1} = d$. We also add inter-layer edges between every two consecutive layers as follows. Between layer $i - 1$ and layer i, we add the edge (r_{i-1}, r_i) for

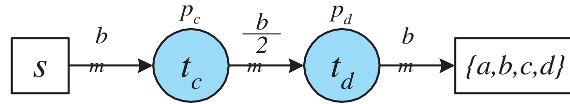


Figure 8: Multicast session graph for a video transfer application

each processing node $r \in R_i$, and apply the scaled cost $c(r) \times p(t_i)$.

The intuition in this formulation is that any path from layer 0 to layer k is forced to include k inter-layer edges, where the ith inter-layer edge corresponds to one of the processing nodes capable of the ith step. Then, the projection of a path to the original graph forms the path that passes through the k processing nodes in the given order.

Therefore, for the unicast session given earlier, we compute the least cost path from s_0 to d_k, where s and d are the source and the destination. The projection of the path on the network graph now forms a legitimate configuration for the session connecting the two terminals and containing the k processing nodes. An example of the layered network for a session with two processing steps t_1 and t_2, the bandwidth requirements $b(s, t_1) = 1, b(t_1, t_2) = 2$ and $b(t_2, d) = 1$, and the processing capacity requirements $p(t_1) = 1$ and $p(t_2) = 3$, is given in Figure 7(a) with the least cost path drawn with thick lines. The projected configuration of the path is also given in Figure 7(b) with its configuration cost.

In fact, the projection of the least cost path is the least cost configuration, which can be proved using a similar argument to the one given for single step processing in Section 3.2.1. Therefore, the layered graph method solves the optimal session configuration problem for unicast sessions with an arbitrary number of processing steps.

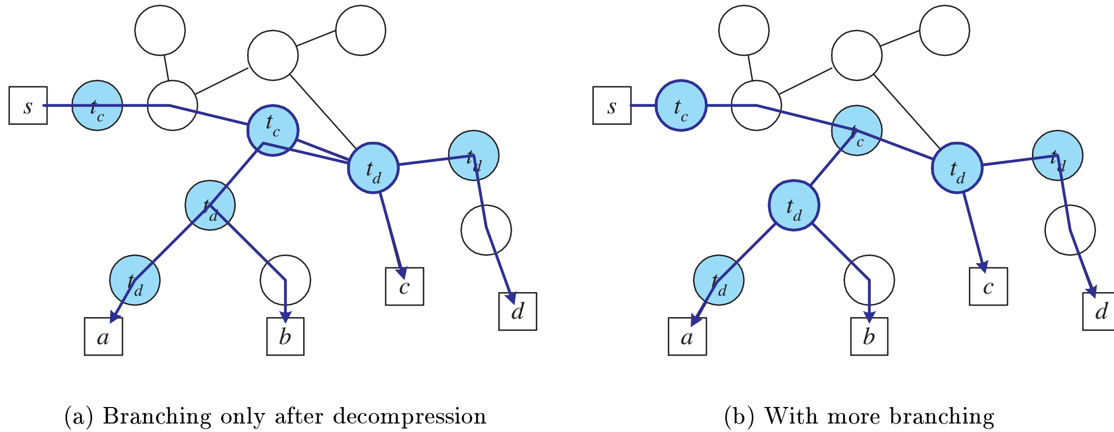
Again, the time complexity is dominated by the shortest path computation. The layered graph contains $(k+1)|V|$ nodes and $(k+1)|E| + \sum_i |R_i|$ edges. The least cost path in the graph can be found in $O((k+1)(|V| \log |V| + |E|) + \sum_i |R_i|)$. Particularly as k grows larger, the layered graph method outperforms the comparison method that was presented in the beginning of this section.

Furthermore, for any given value of k, the layered graph method takes no more than about k times the time it takes to compute the regular least cost path in the network graph and certainly is feasible for dynamic session configurations.

3.3. Configuring single source multicast sessions

In this section, we discuss the configuration of multicast sessions, which involve a single source and multiple receivers. The configuration for such a session should provide a set of link resources that connect the source with each receiver in the session. In the particular form of a multicast, the link resources may be shared among data flows terminating at different receivers, and the configuration often forms a tree shape rooted at the source.

Now, we consider the intermediate processing applied to the data flows of such multicast sessions. As an example, consider a video transfer application that provides a single-source multicast session where the video data coming from the source is compressed to reduce transmission cost and decompressed before it gets to each of the receivers. Figure 8 shows



(a) Branching only after decompression

(b) With more branching

Figure 9: Multicast session configurations

a session graph for this example.

The graph resembles the session graph of the unicast session with two processing steps introduced earlier, with the destination node now representing all receivers in the multicast. The edges in the session graph are of a special type to reflect that link resources can be shared among multiple data flows terminating at different receivers. The session graph also shows the changes in the bandwidth requirements when the compression ratio is 50%. Possible configurations of the session are shown in Figure 9 specified with think lines, each of which forms a tree with different branching points and different choices of processing nodes. The optimal configuration of the session is the one that has the least cost among all possible configurations.

One way to configure the session is to configure each data flow as a unicast session. However, we cannot take advantage of the shared resources with the separate configurations. Instead, we can directly find a multicast configuration in the layered graph, which is constructed in the same way as for unicast sessions. Figure 10 shows the layered graph. To configure the session, find a multicast tree rooted at s_0 and terminating at a_2, b_2, c_2 and d_2 in the layered graph. Then, project it to the original network graph in the same way as for unicast sessions. The projection corresponds to a proper configuration for the multicast session with the processing applied to all data flows. The multicast tree shown in Figure 10 corresponds to Figure 9(b).

Therefore, by finding a configuration in the layered graph, we equivalently configure a multicast session with processing in the network graph. The layered graph method again lets us hide the processing requirement with the graph transformation and solve the problem as a conventional multicast configuration.

On the other hand, finding the least cost configuration is more complex for multicast sessions. In fact, the multicast routing problem is equivalent to the $\mathcal{NP}$-hard Steiner tree problem in graphs.

The Steiner tree problem has been studied extensively for undirected graphs, and there are several approximation methods for it with the best method known having a worst-case

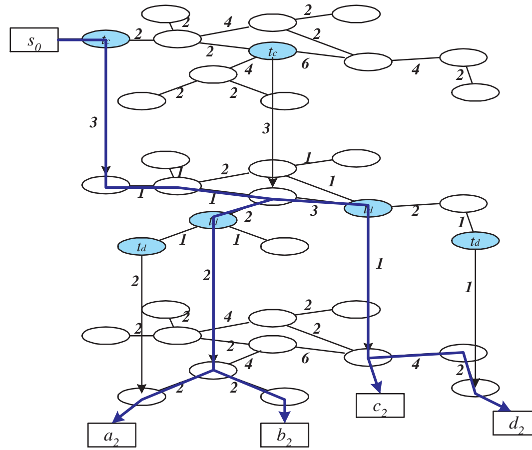


Figure 10: Layered network for multicast

performance ratio of 1.55 [20]. However, those methods cannot be generalized for directed graphs. Only recently, the directed Steiner tree problem has been studied actively and there exist a few approximation methods. The method presented in [19] gives the ratio of 2Ψ in time $O(k^2 b + |E|)$ where Ψ represents the asymmetry of the optimal Steiner tree, and $b(< |V|)$ is a tunable variable in the algorithm, while another set of methods in [5] gives the ratio of $i(i-1)k^{1/i}$ in time $O(|V|^i k^{2i})$, given any $i > 1$, where k is the number of receivers for both methods.

Unfortunately, the results for directed graphs are not appropriate for dynamic session configurations since their time complexity is excessive. We suggest a simple greedy heuristic which attempts to find a good configuration. For the source s and the set M of receiver nodes in the network G, we incrementally construct the multicast tree T, starting with $T = (\{s\}, \{\})$.

Repeat the following step until the set M is empty. Find the receiver $d \in M$ that is nearest to T. The distance from T to d is measured as the shortest path between any node in T to d. Then, augment the tree T to include the shortest path from T to the selected receiver d and remove d from M.

We plan to investigate the efficiency and the performance of this heuristic in programmable networks during the course of this research.

3.4. Configuring sessions with capacity constraints

In this section, we raise an issue in configuring sessions when the network has hard limits on the resource capacities that can be consumed by application sessions. For instance, a video conference application may require a fixed bandwidth available on the links in order to achieve desirable video quality for its interactive sessions, while some link resources may not have enough capacity to accommodate the required bandwidth. Likewise, processing nodes may also be required to have a certain amount of processing capacity in order to handle the target processing while some nodes are lack of the required capacity. In the session configuration problem given in Problem 3.1, however, we did not include capacity

limits in the network graph, and considered every resource in configuring sessions regardless of the available capacity.

In order to explicitly consider the capacity issue, we redefine the session configuration problem with the available capacity at each resource specified in the network graph. The problem statement is given in Problem 3.2.

PROBLEM 3.2 *Session Configuration Problem for Capacity Constrained Networks*
Given: *A directed session graph $G_s = (V_s, E_s)$, with a type $t(u)$ and a capacity requirement $p(u)$ for each vertex $u \in V_s$, and a bandwidth requirement $b(u,v)$ for each edge $(u,v) \in E_s$. Also, a directed network graph $G = (V, E)$ with a type set $T(u)$ and processing capacity $P(u)$ for each vertex $u \in V$, and available bandwidth $B(u,v)$ for each edge $(u,v) \in E$.*
In addition, the unit costs, $c(u)$ and $c(u,v)$, are given for each vertex and edge as a positive integer value.
Find: *A location function $l : V_s \to V$ and a routing function $r : E_s \to 2^E$ that satisfy*

$$\forall u \in V_s, \qquad t(u) \in T(l(u)) \tag{4}$$

$$\forall (u,v) \in E_s, \qquad r(u,v) \text{ is a simple path in } G \text{ from } l(u) \text{ to } l(v) \tag{5}$$

$$\forall x \in V, \qquad \sum_{\substack{u \in V_s: \\ x = l(u)}} p(u) \leq P(x) \tag{6}$$

$$\forall (x,y) \in E, \qquad \sum_{\substack{(u,v) \in E_s: \\ (x,y) \in r(u,v)}} b(u,v) \leq B(x,y) \tag{7}$$

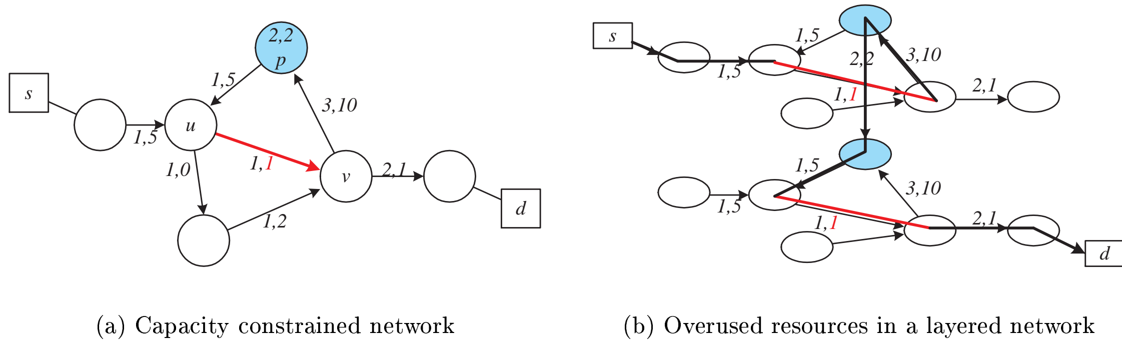
and that minimizes the cost $C_{l,r}$ where

$$C_{l,r} = \textstyle\sum_{u \in V_s} c(l(u))p(u) + \sum_{(u,v) \in E_s} c(r(u,v))b(u,v) \tag{8}$$

This problem extends Problem 3.1 further by adding the capacity constraint of each resource in the network graph while keeping the objective of finding the optimal location and routing functions. Each of Conditions (6) and (7) states that the available capacity should be sufficient to accommodate the total capacity consumption at each resource.

Now, the network graph maintains another variable for each resource to express the available capacity. We refer to this network model as capacity-constrained networks. Given the new network model and the capacity requirement, we now reconsider the optimal configuration problem focusing on unicast sessions.

First, configuring sessions that do not require intermediate processing can be done simply by eliminating the links that lack the required bandwidth from the network graph and finding the least cost path in the "reduced" network. Because the "reduced" network only contains links with enough capacity for the bandwidth requirement and no link is used more than once (in the least cost path), we can guarantee that the optimal configuration computed in this network will always satisfy the capacity requirement.



(a) Capacity constrained network (b) Overused resources in a layered network

Figure 11: Session configuration in capacity constrained networks

On the other hand, the same strategy may not always work for sessions that do require intermediate processing. This is because a single resource may be used multiple times in a selected configuration. An example is given in Figure 11(a), where the session from s to d is to be configured with one processing step. Here, the shaded node is the only processing node capable of the processing, and each resource is associated with two values, the first being its cost and the second being the available capacity. Here, while the session can be configured if 0.5 unit of link bandwidth is required, there is no possible path if 1 unit is required because (u, v) is used twice in the only possible path.

Figure 11(b) shows the corresponding layered graph, in which it is unclear what capacity to assign to each edge. In the figure, we show the least cost path in the layered graph after ignoring links with less than 1 unit of capacity. It is easy to see that the configuration shown is valid if the session requires ≤ 0.5 units of bandwidth but invalid otherwise.

The reason behind these difficulties is that the general problem of configuring sessions with capacity constraints and processing requirements is intractable. Consider a complete network, $G = (V, E)$ where every node except s and d is capable of any type of processing, and each has 1 unit of available capacity. Now, consider a session from s to d which requires $|V| - 2$ processing steps. Any feasible configuration to this problem must pass through all the intermediate nodes, and thus it provides a solution to the well-known Hamiltonian path problem [9], which is known to be $\mathcal{NP}$-complete.

3.5. Heuristics for configuring sessions with capacity requirements

In this section, we introduce two heuristic methods for the optimal session configuration problem in a capacity-constrained network. We focus on unicast sessions that specify a processing requirement for each step and a bandwidth requirement for each path segment between two consecutive steps. Note that the bandwidth requirement can differ on different path segments, since processing steps may expand or reduce the amount of data. As described earlier, the costs in the different layers are scaled to account for such effects. Our heuristics extend the layered graph method to prevent resources from being used beyond their current capacity.

The first heuristic is really a collection of similar algorithms, that we refer to as *selective*

edge inclusion algorithms. Each modifies the layered graph to prevent links from being over-used and then finds a shortest path in the modified graph. The algorithms differ in the way they modify the layered graph.

- The *strict inclusion* method includes an edge in the layered graph only if it has enough available capacity so that it cannot be over-used, even if it is selected for use in all layers. This policy applies to both intra-layer and inter-layer edges. Since different processing steps may require different amounts of processing capacity, we include a given edge as an inter-layer edge only if the sum of the capacities required for all the processing steps is no larger than the available capacity of the processing node represented by the inter-layer edges. Similarly, we include a given intra-layer edge only if its capacity is no smaller than the sum of the bandwidth requirements for all path segments. Once the modified layered network is constructed, a shortest path from the source to the destination is found. If none exists, the session configuration attempt is rejected.
- The *loose inclusion* method includes an edge in all layers if it has sufficient capacity to be used in any layer. If, after a path is determined, the path is found to over-use some edge, the path is discarded and the session configuration attempt is rejected.
- The *permissive loose inclusion* method is not intended as a practical algorithm, but is used in the simulation study described below to provide a nominal bound on the performance of the other algorithms. It works like the loose inclusion method, except that it never rejects the path that is found, even if the path over-uses some edge.
- The *random inclusion* method includes edges in a set of selected layers for which the total capacity is no larger than the edge capacity. For each edge, the layers are selected randomly and independently. Once the modified layered network is constructed, shortest path search is done. If successful, the session is configured using that path.
- The *consecutive inclusion* method picks a layer at random and then goes through the remaining layers in consecutive order, adding the edge to each layer in which the addition does not violate the capacity constraint.

The selective edge inclusion methods are very simple to implement and can perform reasonably well when the session resource requirements are much smaller than the capacities of the links and processing nodes.

Our second heuristic is somewhat more complex but can perform well, even when session resource requirements are relatively large. The algorithm is an extension to Dijkstra's shortest path algorithm, and is called the *capacity tracking* algorithm. We start with a brief review of Dijstra's shortest path algorithm.

Given a graph, and a source node s, Dijkstra's algorithm computes a *shortest path tree* rooted at s. Initially, the tree contains just s. The algorithm maintains a set S, of *boundary vertices*, which includes all nodes v that are connected to a vertex u in the partial tree constructed so far, by a directed edge (u, v). At the start of the algorithm, S contains the nodes v, for which there is an edge of the form (s, v). The algorithm also maintains, for each vertex v, a *tentative distance* $d(v)$, which is the length of the shortest path from s to v that

has been found so far. It also maintains a *tentative parent* $p(v)$, which is the predecessor of v in a path from s of length $d(v)$. The quantities $d(v)$ and $p(v)$ are not defined for nodes that are neither in the tree, nor in S.

At each step, Dijkstra's algorithm selects a node v in S for which $d(v)$ is minimum, and adds it to the tree. It then examines each edge (v, w). For each node w that is neither in the tree nor in S, it adds w to S, setting $p(w)$ to v and $d(w)$ to $d(v)$ plus the length of (v, w). For each node w that is in S, it compares $d(w)$ to $d(v, w)$ plus the length of the edge (v, w), and if it finds that $d(w)$ is larger, it updates $d(w)$ and $p(w)$. If the set of boundary vertices is implemented using a Fibonacci heap [6], Dijkstra's algorithm runs in $O(m+n \log n)$ time, where n is the number of nodes in the graph, and m is the number of edges.

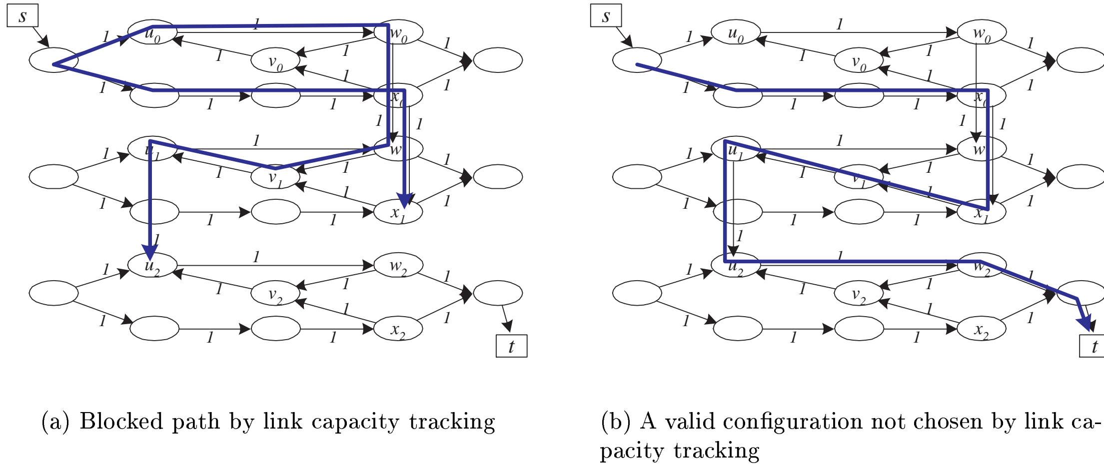
When Dijkstra's algorithm is applied to a layered graph, some of the paths in the shortest path tree may contain edges on different layers that correspond to the same link or router in the original network, from which the layered network was constructed. This may lead to over-use of resources. To prevent this, we modify the basic processing step, to include a check for over-used resources. In particular, when a node v_i is added to the tree (i denotes the layer in which the vertex appears), we consider edges of the form (v_i, w_i) and (v_i, v_{i+1}). Before processing an edge of the form (v_i, w_i), we examine the path in the tree from s to v_i and add up the capacities required by all edges on the path that correspond to the original link (v, w). If this total capacity, plus the capacity that would be used by the edge (v_i, w_i) exceeds the available capacity of the link, then no action is taken with respect to that edge. Edges of the form (v_i, v_{i+1}) are handled similarly. We refer to this capacity checking procedure as *link capacity tracking*.

The extra time required by link capacity tracking is $O((km)(kn))$, in the worst-case, where m and n are the number of edges and nodes in the original network and k is the number of processing steps. This can be seen by noting that the checking procedure is invoked no more than $k(m+n)$ times and each execution requires that we traverse a path with no more than $kn-1$ edges.

The running time can be improved by maintaining an additional variable $\kappa(v_i)$ for each vertex in the partial tree constructed so far. If $u_i = p(v_i)$, then $\kappa(v_i)$ is the sum of the capacities required from all edges on the tree path from s to v_i that are copies of the link (u, v) in the original network graph. Similarly, if $v_{i-1} = p(v_i)$, then $\kappa(v_i)$ is the sum of the capacities required from all edges on the tree path from s to v_i that correspond to the processing node v in the original network graph. Using these additional variables, we can terminate the capacity tracking search from a node v_i back to s early, reducing the time taken for capacity tracking to $O(kmn)$.

In practice, the extra time required by capacity tracking is much smaller than the worst-case analysis suggests, because networks are designed to have small diameter, which means that the paths in the shortest path tree generally have far fewer than kn edges. If we let D denote the maximum number of edges in a path from the root to a vertex in the shortest path tree, then the extra time required by link capacity tracking is $O(kmD)$. Even this result over-states the time required by capacity tracking in practice. As will be seen later, running time measurements show that capacity tracking takes less than double the time required by the simpler heuristics, in more realistic situations.

Link capacity tracking ensures that paths found by the algorithm do not over-use any



(a) Blocked path by link capacity tracking

(b) A valid configuration not chosen by link capacity tracking

Figure 12: Blocked path in Link capacity tracking

resources. However, since the problem is $\mathcal{NP}$-hard, we cannot expect it to always find a valid path, even when a path exists. Consider the example shown in Figure 12(a). If each link in the original network graph has one unit of capacity and the session requires one unit of capacity on each edge of the selected path, it can fail to find a path, as shown in part (a) of the figure. The bold edges are the edges that form the shortest path tree, at the time the path search terminates. Note that there is no way to extend the tree further, since the only edge leaving vertex u_2 has already been used in the top layer, and hence cannot be used again. On the other hand, there is a path that could be used for this session, as shown in part (b).

We performed a set of simulations for the session configuration problem to evaluate the heuristic methods discussed so far. In the simulations, we considered the following four different network topologies.

- *Torus*: This network is based on a grid of 64 nodes where every node has an outgoing edge to each of its four neighbors, north, south, east and west along the grid lines. The nodes at edges of the square grid also have links that "wrap around" to the corresponding node at the opposite edge, resulting in a torus topology. Figure 13 shows the network topology. Nodes that can perform processing are shown as triangles.

- *Random*: This network is a random regular network with 64 nodes, each having 4 incident edges. We build the network starting with a random degree-bounded tree that spans all 64 nodes, then we expand the network by adding edges randomly until every node has exactly four incident edges.

In both networks, every link has the same capacity and the same cost, and one third of the nodes are randomly designated as processing nodes, with the ability to perform processing. All processing nodes have the same capacity.

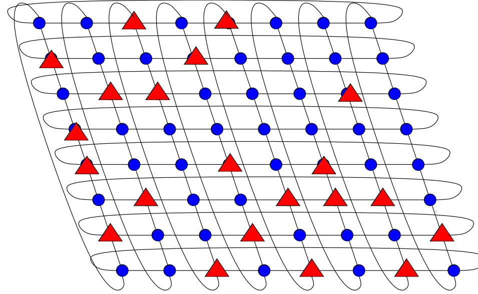


Figure 13: Torus network

- *Metro 20*: This network is a more realistic network configuration, spanning the 20 largest metropolitan areas in the United States. The network topology is shown in Figure 14(a). Nodes that are capable of performing processing are shown as triangles. Link costs are set equal to the physical distance between the nodes they connect, reflecting the higher cost associated with links spanning greater distances. The link capacities are selected to be large enough to handle the anticipated traffic. The link dimensioning procedure used for this purpose is taken from [16], which describes a constraint-based network design methodology and an interactive network design tool that implements it. We constrain the traffic in two ways. First, the total traffic entering and leaving a node is chosen to be proportional to the population of the metropolitan area represented by that node. Next, for each node u, we constrain its traffic to every other node using constraints that are proportional to the populations of the metropolitan areas represented by the other nodes. Specifically, if δ_v is the fraction of the population outside node u, that is associated with node v, then we limit the traffic between u and v to be no more than $1.3\delta_v$ times the total traffic entering and leaving node u. The factor of 1.3, was chosen to allow for some flexibility in the distribution of traffic, reflecting the natural variations that occur in network traffic. Given these traffic assumptions and a *default path* joining each pair of vertices, link dimensions can be computed using linear programming. The resulting link capacities guarantee that any traffic pattern satisfying the traffic constraints can be carried if the traffic is routed along the default paths. The default path between a pair of vertices is a shortest path containing at least one processing node, and can be found using a two layer network. The processing nodes along each default path are dimensioned to handle the worst-case traffic load allowed by the traffic constraints. When performing the simulations, we do not constrain the traffic to use just the default paths, but the link dimensions are chosen, under the assumption that the default paths are used.

- *Metro 50*: This network is a larger version of the *Metro 20* network. It has a node for each of the fifty largest metropolitan areas in US. The topology is shown in Figure 14(b). The links and processing nodes are dimensioned in the same way as *Metro 20*.

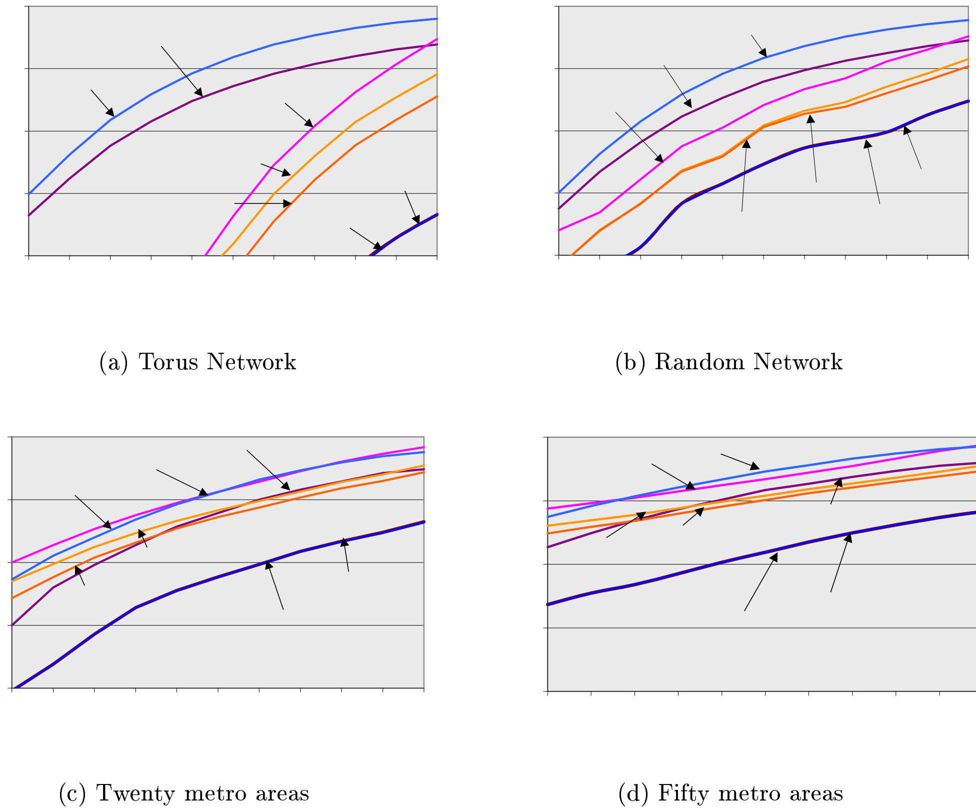
On the four network topologies, we simulated the heuristics for over 2.5 million sessions with three processing steps and the capacity requirement equal to 3% of the average link bandwidth.

For each configuration attempt, we selected the end points randomly for *Random*, while

(a) Metro 20 Network

(b) Metro 50 Network

Figure 14: Metro Areas



(a) Torus Network

(b) Random Network

(c) Twenty metro areas

(d) Fifty metro areas

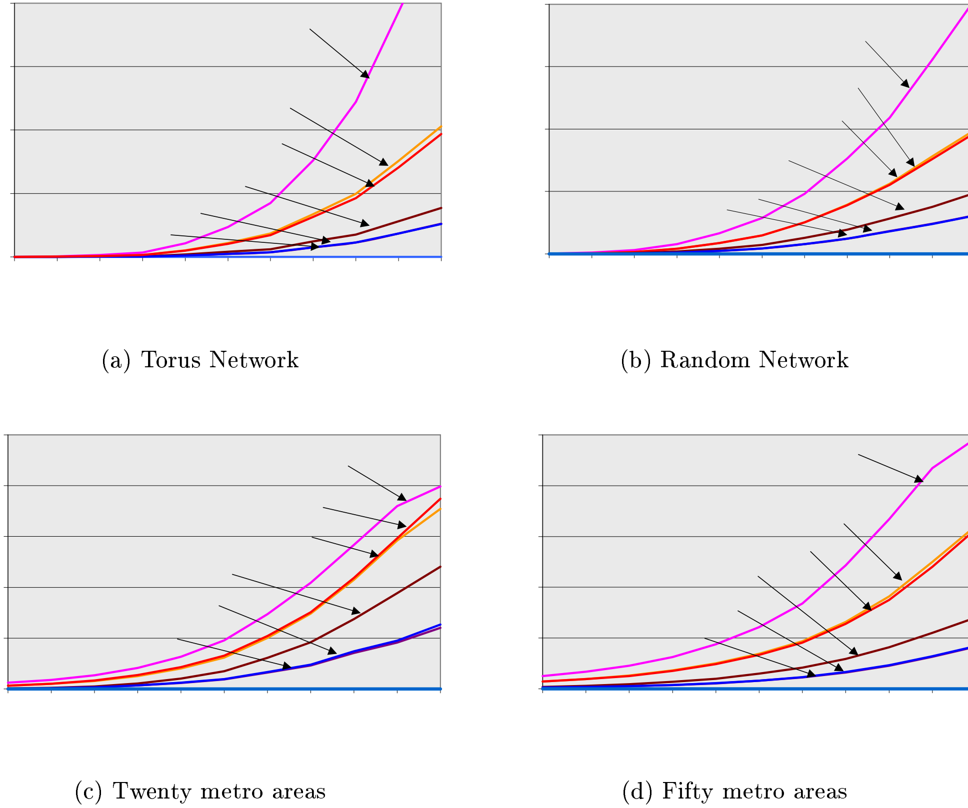
Figure 15: Heuristics for session configurations

restricting them to be exactly four hops apart for *Torus*. For *Metro 20* and *Metro 50*, the selection of the end nodes was weighted by the populations of the metro areas, reflecting the higher traffic volumes expected in larger cities.

The performance of the heuristics is measured with the blocking probability, which is the percentage of session configuration attempts that were unsuccessful. Figure 15 shows the blocking probabilities for the various heuristics, as a function of the offered load. The plots also show the blocking probability when sessions are constrained to use a fixed default path, which is initialized to the least cost path in the layered graph when capacity is considered unlimited. Among the network topologies, *Torus* shows the best performance due to abundant connectivities between any end point pair. However, without being able to take alternate paths, the default path always shows the worst blocking probabilities that are somewhat consistent in all topologies. *Loose Inclusion* is also blocked with high probability by overusing resources.

For other topologies, all heuristics show higher blocking probabilities while *Link Capacity Tracking* performs almost as good as the lower bound with blocking probabilities of less than 1% at offered loads of more than 75%.

We also measured the cost of the successful configurations. In Figure 16, we show the configuration cost from all heuristics relative to the cost of the default shortest path, which



(a) Torus Network

(b) Random Network

(c) Twenty metro areas

(d) Fifty metro areas

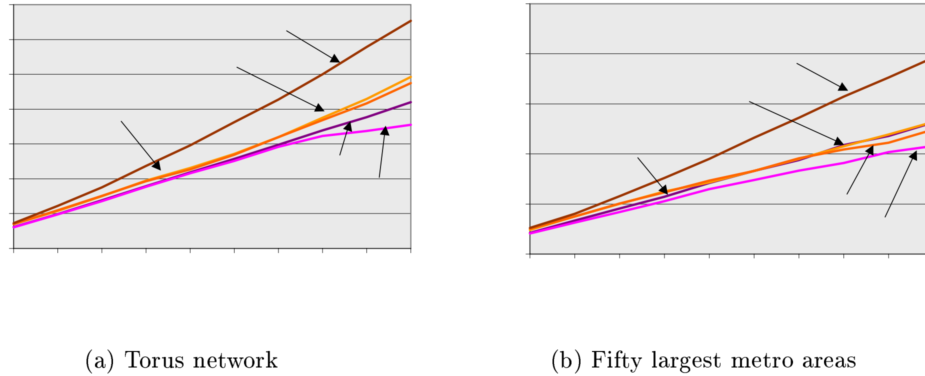
Figure 16: Configuration Cost

is a lower bound. All heuristics provide nearly optimal costs at low loads, but deviate significantly at higher loads. The paths produced using *Link Capacity Tracking* generally stay within 5 to 10% of the lower bound up to loads of 95%.

Lastly, we measured the average time required for session configuration by the different algorithms. Figure 17 shows the results for *Torus* and *Metro 50*. For all algorithms, we varied the number of steps from 1 to 10. As can be seen, the algorithms based on selective link inclusions are the fastest. On the other hand, link capacity tracking remains reasonably competitive, with a computational cost less than twice that of the best selective inclusion algorithm when ten processing steps are performed. Considering that sessions are likely to have far fewer than 10 steps in the vast majority of applications, the superior blocking probability achieved with *Link Capacity Tracking* more than compensates for the extra computational time.

4. Network Design in Programmable Networks

In the previous section, we introduced the concept of programmable routers, equipped with the capability to dynamically install and run processing modules. In the session



(a) Torus network

(b) Fifty largest metro areas

Figure 17: Time requirements for session configurations

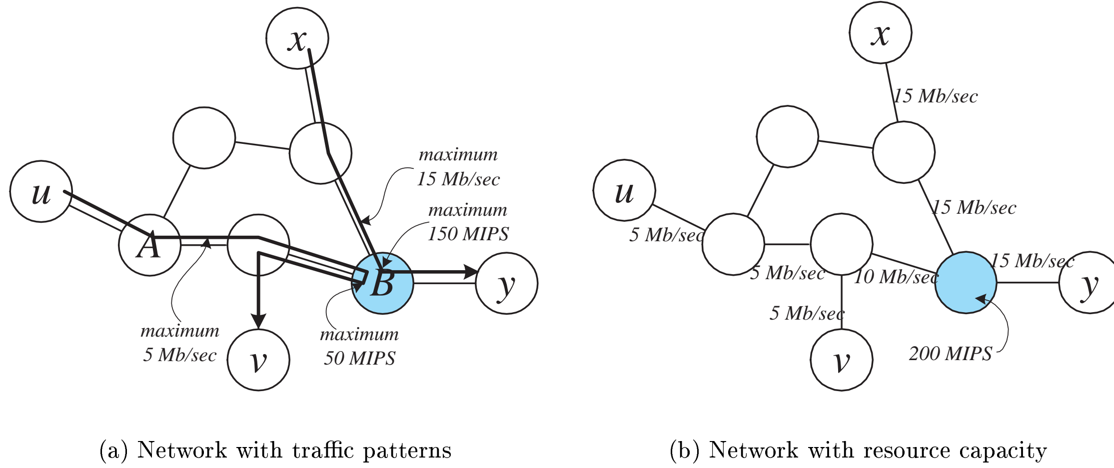
configuration problem, our objective was to find the optimal (the most cost effective) set of resources to support each session, where the resources included the links and the processing nodes required by the session.

In this section, we address the problem of how to provision a programmable network to ensure that it can carry an anticipated traffic load with good performance, while achieving efficient resource usage.

In conventional networks, links are the major resources shared by application sessions. In the "link dimensioning problem" in conventional networks, the goal is to dimension all links in the network so that any possible set of application sessions can be configured without blocking. Now, as we allow intermediate processing, sessions often take a set of links that are different from the ones used in conventional networks (where no processing is required), and thus require links to be dimensioned differently. In addition, processing nodes also need be dimensioned.

Let us consider the example given in Figure 18(a). We assume that the maximum traffic for sessions with a processing requirement is expected to be 5 Mb/sec from the end node u to node v, 15 Mb/sec from x to y, and zero between all other nodes. Also, suppose that we require sufficient processing capacity to execute 10 instructions per bit of data sent. The processing limit at the processing node (shaded) is adjusted for the maximum traffic that goes through the node and shown in units of *MIPS* in the figure (1 *MIP* = 1 million instruction per second). Our objective is to provision the network so that any collection of sessions satisfying these overall constraints can be supported. Specifically, we must determine how much capacity to assign to each resource on the paths in Figure 18(a). One solution is shown in Figure 18(b). While we require the network to be nonblocking for the given traffic constraints, we naturally prefer not to assign more resources than are needed. So, we associate each link and each processing node with a cost per unit capacity. Among all the possible network configurations that satisfy the given traffic constraints, we seek the one with the smallest overall cost.

The resource dimensioning problem has been studied in the context of conventional networks [14][21] which do not support intermediate processing. In this proposal, we discuss



(a) Network with traffic patterns

(b) Network with resource capacity

Figure 18: Dimensioning Resources

the problem of dimensioning a programmable network for a general set of traffic for sessions that do require intermediate processing.

We also consider the problem of how to choose the locations for processing resources. We discuss how the processing locations affect the overall network cost and propose methods for determining optimal or near optimal locations.

4.1. Dimensioning resource capacity

In this section, we discuss the problem of dimensioning resources that are shared among application sessions in a programmable network . For the moment, we focus on the class of unicast applications whose sessions require a single processing step. We denote such sessions as a tuple (s, b_1, p, b_2, t), which represents a unicast path from the source s to the destination t that passes through a node with processing capacity p. In the tuple, we also specify the link bandwidth b_1 for the path segment from s to the processing node and b_2 for the path segment from the processing node to t. The target application class includes any unicast applications with this session format where the value of b_1 and b_2 can be varied.

We assume that the network topology and the processing sites are given. We also assume that all traffic between a pair of nodes is carried on a given fixed path (typically the shortest path). Our goal is to determine the capacity of each link and each processing node in order to satisfy the application sessions.

Following [8], we have two types of traffic constraints; termination constraints and pair-wise constraints. For each node u, we let $\alpha(u)$ be the maximum session bandwidth that originates from u (source limit), and $\omega(u)$ be the maximum session bandwidth that terminates at u (sink limit). For each pair of nodes, u, v, we also have a constraint $\delta(u, v)$ on the traffic from u to v.

Given traffic constraints $\Gamma = (\alpha, \omega, \delta)$, we say that a session (s, b_1, p, b_2, t) is "compatible" if it satisfies the traffic limits, i.e. $b_1 \leq \alpha(s), b_2 \leq \omega(t)$, $b_1 \leq \delta(s, t)$, and $b_2 \leq \delta(s, t)$.

Furthermore, we say that a set of sessions S is "compatible" if the total traffic satisfies Γ. In other words, for all s and t,

$$\sum_{\substack{(s',b'_1,p',b'_2,t')\in S\\ s'=s}} b'_1 \le \alpha(s), \qquad \sum_{\substack{(s',b'_1,p',b'_2,t')\in S\\ t'=t}} b'_2 \le \omega(t)$$

and

$$\sum_{\substack{(s',b'_1,p',b'_2,t')\in S\\ s'=s,t'=t}} b'_1 \le \delta(s,t), \qquad \sum_{\substack{(s',b'_1,p',b'_2,t')\in S\\ s'=s,t'=t}} b'_2 \le \delta(s,t)$$

The formal statement of the problem appears below.

Problem 4.1 *Resource Dimensioning Problem in Programmable Networks*
Given: *A network graph $G = (V, E)$ with a cost $c(l)$ for each link $l \in E$, a set $R \subseteq V$ of processing nodes with a cost $c(w)$ for each node w, and a set of traffic constraints $\Gamma(\alpha, \omega, \delta)$. Also, for each pair of nodes u, v, a path $\pi(u, v)$ which passes through a processing node $r(u, v) \in R$.*
Find: *Link capacities $cap(l)$, for all links $l \in E$, processing capacities $cap(w)$ for all nodes $w \in R$ that satisfy*

$$cap(l) \ge \sum_{\substack{(s,b_1,p,b_2,t)\in S\\ l\in\pi_1(s,t)}} b_1 + \sum_{\substack{(s,b_1,p,b_2,t)\in S\\ l\in\pi_2(s,t)}} b_2 \tag{9}$$

$$cap(w) \ge \sum_{\substack{(s,b_1,p,b_2,t)\in S\\ w=r(s,t)}} p \tag{10}$$

for all links l, all nodes w and all sets of sessions S that are compatible with Γ, and that minimizes

$$D(G, R, \Gamma) = \sum_{l\in E} cap(l)cost(l) + \sum_{w\in R} cap(w)cost(w) \tag{11}$$

Conditions (9) and (10) state that the capacity assigned to each resource is large enough to accommodate any set of sessions satisfying Γ. Due to the processing requirement, each session path $\pi(u, v)$ is composed of two path segments, $u \sim r(u, v)$ and $r(u, v) \sim v$, that may overlap with each other. We denote the first path segment as $\pi_1(u, v)$ and the second path segment as $\pi_2(u, v)$. Since a link may appear in more than one path segment with a different capacity requirement, we considered the two cases separately and combine them to bound the total capacity in (9).

In programmable networks, resource dimensioning is complicated by two factors that are not present in the conventional resource dimensioning. First, the goal of the problem is expanded to dimensioning processing resources in addition to link resources. Second, while in conventional networks unicast sessions are configured on simple paths (no repeated links or nodes), in programmable networks the best end-to-end path may not be simple.

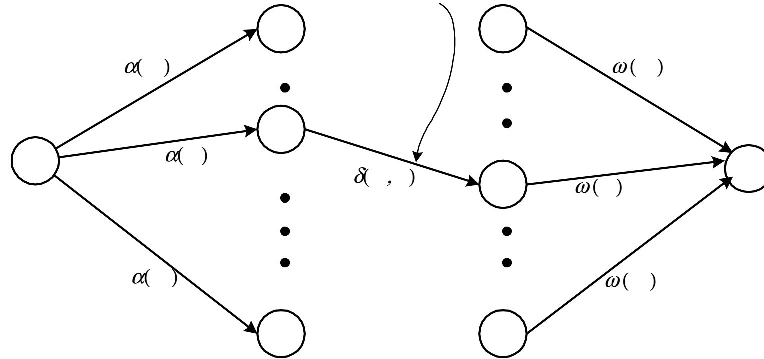


Figure 19: An instance of the max flow problem for dimensioning (u, v)

Nonetheless, we found that a method used for dimensioning links in conventional networks can be generalized for programmable networks.

The problem of link dimensioning in conventional network is discussed in [8], where Fingerhut asserts that the capacity assigned to each link by the minimum cost dimensioning is actually the minimum capacity required at the link to accommodate any compatible set of sessions. He then formulates the problem of finding the minimum capacity at each link as an instance of the *max flow problem*.

The general max flow problem [6] consists of a directed graph, edges labeled with capacity bounds, and two distinct nodes; the source and the sink. In this graph, a flow is an assignment of values to each edge, such that the value does not exceed the capacity bound at each edge, and the sum of the incoming flows equals to the sum of the outgoing flows at each node except at the source and the sink. The goal of the max flow problem is to find a flow in the network which maximizes the incoming flow at the sink.

Figure 19 shows the instance of the *max flow problem* for computing the minimum capacity required at link (u, v) in a conventional network. This max flow instance contains a source node s_u that corresponds to u and a sink node t_v that corresponds to v. The graph also contains two columns of copies of all nodes in the network. Each copy in the first column, denoted w_i^s, is connected from s_u with an edge of the capacity bound $\alpha(w_i)$, while the copy in the second column, denoted w_i^t, is connected to t_u with an edge of the capacity bound $\omega(w_i)$. We complete the graph by including an edge for each pair of node (w_i^s, w_j^t) only if fixed path $\pi(w_i, w_j)$ for sessions from w_i to w_j goes through the link (u, v). The capacity of this edge is set to the pairwise traffic limit $\delta(w_i, w_j)$. In this formulation, the value of any flow is a permitted amount of traffic at (u, v) given the traffic limits, $\Gamma = (\alpha, \omega, \delta)$. Therefore, by finding the maximum flow in this graph, we can obtain the maximum amount of traffic that can be observed at (u, v), and thus the minimum amount of capacity needed at (u, v) to handle any set of sessions satisfying the traffic limits.

Now, we generalize the max flow formulation to dimension the same link (u, v) in a programmable network. Constructing the max flow graph is done in the same way as for conventional networks. This formulation, however, does not present the actual traffic limit at the link (u, v) in programmable networks because each configuration path $\pi(w_i, w_j)$ may

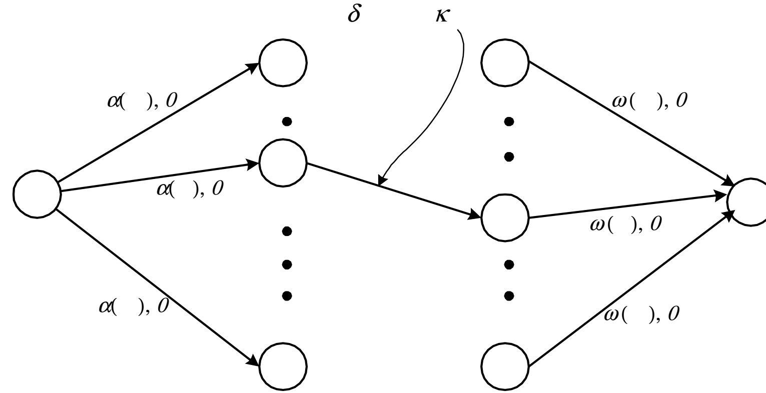


Figure 20: An instance of the max cost max flow problem in programmable networks

use the link multiple times. For example, although the maximum traffic limit between w_i and w_j is the minimum of $\alpha(w_i)$, $\omega(w_j)$, and $\delta(w_i, w_j)$), the actual limit observed at the link (u, v) should be scaled up by $\kappa(w_i, w_j, (u, v))$ i.e. $min(\alpha(w_i), \omega(w_j), \delta(w_i, w_j)) \times \kappa(w_i, w_j, (u, v),$ where the quantity $\kappa(w_i, w_j, (u, v))$ is the number of times that (u, v) appears on the path $\pi(w_i, w_j)$.

To represent this effect, we apply to each edge (w_i^s, w_j^t) the quantity $\kappa(w_i, w_j, (u, v))$ as its cost while applying zero cost to all other edges. Figure 20 shows the formulation. The max flow formulation augmented with edge costs specifies another problem called the *max cost max flow problem*. Here, the cost of a flow at each edge is defined as the product of the flow and the cost, and the cost of a flow in the entire network as the sum of the products.

In our formulation of the max cost max flow problem, because the quantity $\kappa(w_i, w_j, (u, v))$ is given as the cost of the edge (w_i^s, w_j^t), the cost of any legitimate flow in the graph represent the actual traffic observed at the link (u, v). Therefore, the maximum flow from s_u to t_v that also has the maximum cost corresponds to the maximum traffic that would observed at (u, v), thus the minimum amount of capacity needed at (u, v).

The capacity at a processing node x can be found with the same formulation. First, we construct the max flow instance with a new source s_x and a sink t_x replacing s_u and t_v. This time, we include an edge (w_i^s, w_j^t) with a capacity bound $\delta(w_i, w_j)$ only if the processing node x is used by the sessions with end points w_i and w_j, i.e. $r(w_i, w_j) = x$. We also apply to each edge (w_i^s, w_j^t) the cost of $\kappa(w_i, w_j, x)$, which is the number of times x is used as a processing node for sessions between w_i and w_j. The maximum flow with the maximum cost in this graph corresponds to the minimum processing capacity needed at x. Although this capacity requirement is obtained in terms of data traffic (link bandwidth), we can convert it into a processing capacity such as *MIPS* by applying the number of instructions required per unit data bandwidth. The maximum flow instances for all links and all processing nodes determine the minimum capacities needed for all resources in a programmable network, and furthermore yielding the minimum cost dimensioning.

4.1.1. Issues. In this section, we discuss some of the issues in applying the resource dimensioning problem for applications with diverse parameters.

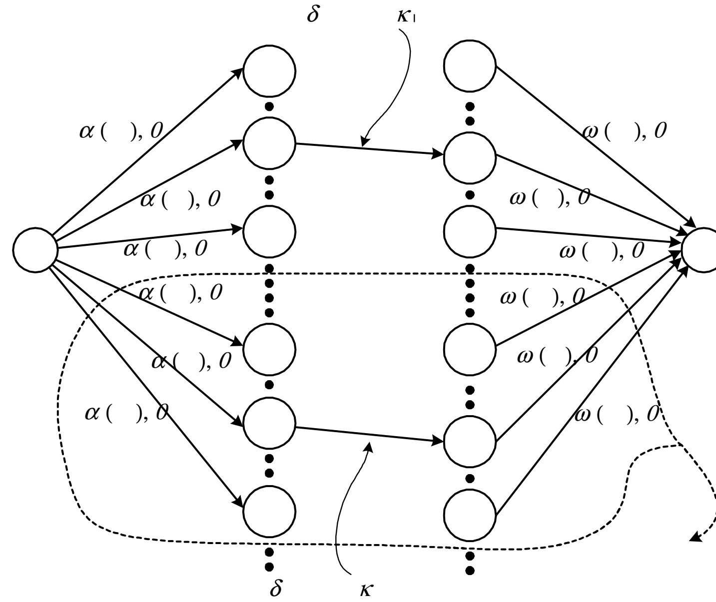


Figure 21: An instance of the max cost max flow problem for multiple applications

Previously in the dimensioning problem, we only considered sessions with one processing step. Now, we further generalize the problem for sessions with an arbitrary number of processing steps, where sessions are denoted $(s, b_1, p_1, b_2, ..., b_k, p_k, b_{k+1}, t)$. Such sessions are configured on a path $\pi(s,t)$ that includes k processing nodes, and therefore is composed of $k+1$ path segments, where a link may appear in any of the $k+1$ path segments.

The max cost max flow formulation for the case of single step processing is general enough to be applied to the case of processing with an arbitrary number of steps. Note that the cost of a flow still represents the correct amount of traffic observed at the target link (u,v) when the quantity $\kappa(w_i, w_j, (u,v))$ is assigned to each edge (w_i^s, w_j^t) as the cost. The argument can be applied for dimensioning processing nodes.

Now that we can handle applications with various session formats, we consider a more realistic situation where the network is shared among applications with different session formats. In this case, the network needs be dimensioned so that any mixture of sessions with different formats satisfying the traffic constraints can be configured without blocking.

As complicated as it might sound to dimension the network for heterogeneous applications, the dimensioning at each resource can be done easily by combining the capacity requirements from all applications. Let us assume that there are total m classes of applications, A_1, A_2,...,A_m, each of which specifies the paths and the processing nodes to be used for its sessions. Also, suppose that we know how much does each application class contribute to the traffic limits. For instance, we have the source traffic limit $\alpha_i(u,v)$ which is the maximum traffic limit from u belonging to the application class i. We denote other traffic limits similarly as $\delta_i(u)$ and $\omega_i(u)$. Naturally, the total traffic limits are given as the

sum of the traffic limits for all applications, i.e.

$$\sum_{i=1..m} \alpha_i(u) = \alpha(u), \sum_{i=1..m} \omega_i(u) = \omega(u), \sum_{i=1..m} \delta_i(u,v) = \delta(u,v).$$

In order to determine the maximum traffic given the m different application classes, we formulate an instance of the max cost max flow problem that combines the instances for all classes. Figure 21 shows the formulation, where m individual max cost max flow instances share the source s_u and the sink t_v. The instance for the application class h is shown inside the dotted line. The value of the max cost max flow in this formulation now corresponds to the maximum traffic that can be expected at (u, v) from all applications, and therefore is the minimum amount of capacity required at (u, v). Dimensioning a processing node x can also be done in the same way.

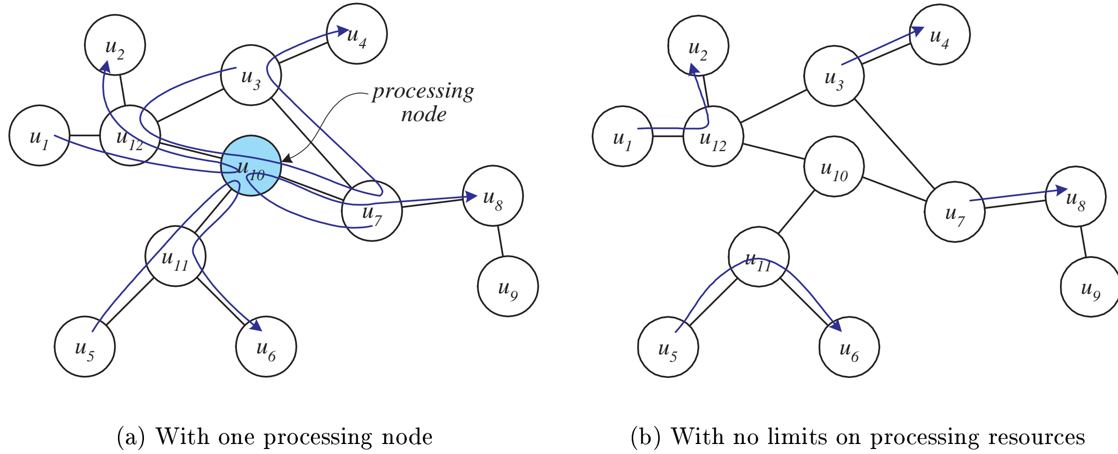
Therefore, by combining the max cost max flow formulation, we can solve the dimensioning problem for an application class with any number of processing steps and also for any combination of application classes in programmable networks. As a special case, we may also dimension networks that have both the conventional traffic (that requires no intermediate processing) and the traffic that requires processing by treating the conventional traffic as simply another application class.

4.2. Placing Processing Resources

We have discussed how to determine the capacity of each resource in programmable networks where the path and the processing nodes were fixed for every pair of end points. In this section, we consider a more generalized situation where we have the freedom to select the nodes for placing processing resources. Our goal is to obtain a least cost dimensioning by carefully placing the processing resources. For the moment, we focus on the case of sessions with a single processing step.

One extreme way to place processing resources is to allow only one processing node in the entire network. In this case, once the processing node is selected, say x, we can identify the path π for every pair of end nodes, which is the shortest path between them that includes x. An example for this case is shown in Figure 22(a), where possible pairs are (u_1, u_2), (u_3, u_4), (u_5, u_6), (u_7, u_8) and u_{10} is the processing node. Now, we can dimension the network as we did in the previous section. To get the least cost dimensioning, we can compute the dimensioning cost for each node as the processing location and select the one that gives the least cost. This is the optimal dimensioning when we limit the number of processing nodes to 1, and in fact is an upper bound for the cost associated with any number of processing nodes.

Another extreme way is not to limit the number of processing nodes at all. That is, let the set R include every node in V, in which case $p(u, v) = p(u, v, R)$, where $p(u, v)$ is the direct shortest path between u and v, and $p(u, v, R)$ is the shortest path that includes any node in R. Figure 22(b) shows the new configurations given in the same network with unlimited processing resources. Here, since every node on the path π can be used for processing, we must identify the node that is actually used for each end point pair to properly dimension resources. However, selecting a set of processing nodes that result in



(a) With one processing node

(b) With no limits on processing resources

Figure 22: Configurations with different processing resources

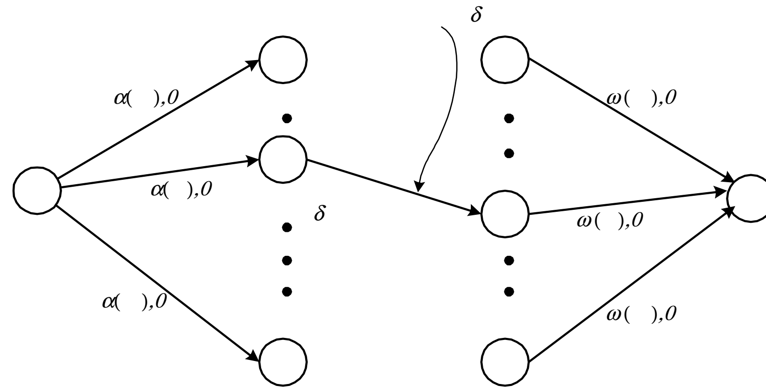


Figure 23: max cost max flow for a lower bound on network dimensioning

the smallest dimensioning cost is a complex task, and is in fact $\mathcal{NP}$-hard given any bound on the number of processing nodes. We will detail this problem in the next section.

In the mean time, we can provide a lower bound on the dimensioning cost in programmable networks with a lower bound in conventional networks because the latter bounds the cost associated with dimensioning links only, and processing requirements always add more costs into the total cost.

A lower bound for conventional network dimensioning is provided in [8] for any network topology given traffic limits Γ, with the assumption that the link costs satisfy the triangle inequality. According to the definition in Problem 4.1, given a network topology, the dimensioning cost must be at least as much as the cost for configuring the most expensive sessions that satisfy the traffic limits Γ. Therefore, when the sessions are configured in the least cost way, which is through the link directly connecting the end points due to the triangular inequality, the configuration cost of the most expensive sessions gives a lower bound to the dimensioning cost for any network topology. We can obtain this lower bound as follows.

First, consider the maximum session traffic compatible with $\Gamma = (\alpha, \omega, \delta)$, which is the max flow of the formulation given in Figure 23, where the first number associate with each edge is the capacity bound. Among the collections of sessions that have the maximum traffic, find the one that is most expensive to configure assuming the direct link configurations with the following instance of the *max cost max flow* problem [6].

In the above max flow formulation, define the cost per unit flow for each edge (w_i^s, w_j^t) as the cost of the direct link $c(w_i, w_j)$ and zero for all other edges. In Figure 23, the second number associated with each edge is the cost per unit flow. Given a flow, the cost at each edge is defined as the product of its cost per unit flow and the flow assigned to the edge, and the cost of the entire flow is the sum of the costs of all edges. The cost of the entire flow then corresponds to the configuration cost for the session traffic specified by the flow. Therefore, by finding the max flow that also has the maximum cost, we obtain the cost for configuring the most expensive session traffic, which is a lower bound to the conventional link dimensioning for any network topology.

Now that we have discussed the measures (the lower bound and the upper bound) which can be used to evaluate our choices of processing nodes and the resulting network dimensioning, we return to the problem of placing processing resources and discuss two properties that are essential in practical situations. First, we assume a budget for processing resources, which allows us to have at most a constant number(K) of processing nodes. Second, we restrict the paths that may be used for each session so that they do not overuse resources. Note that the path $p(u, v, R)$ may deviate from the direct shortest path $p(u, v)$ using additional link resources. We would like to place the processing resources so that no session must take too long a "detour" to reach a processing node. To achieve this objective, we require

$$length(p(u, v, R)) \leq length(p(u, v)) + max(L_{min}, \epsilon \times length(p(u, v)))$$

where L_{min} and ϵ are parameters.

Given the restrictions, we formally state the problem of placing processing resources below.

PROBLEM 4.2 *Processing Placement Problem*
Given: *A network graph $G = (V, E)$, a positive value $\epsilon, L_{min} \geq 0$, traffic limits $\Gamma = (\alpha, \omega, \delta)$, and a positive integer $K \leq |V|$*
Find: *a set $R \subseteq V$ with at most K elements that minimizes $D(G, R, \Gamma)$ such that for every u, v,*

$$length(p(u, v, R)) \quad \leq length(p(u, v)) + max(L_{min}, \epsilon \times length(p(u, v))) \qquad (12)$$

where $p(u, v, R)$ is the shortest path that includes any node in R.

The objective of Problem 4.2 is to find K nodes in the network such that the dimensioning cost $D(G, R, \Gamma)$ is minimized when they are provided as the processing nodes R in Problem 4.1.

Finding the exact solution to the problem is a complex task. In fact, it is proved to be an intractable problem, to which the K-median problem, particularly in 2D Euclidean

space, can be reduced. In brief, the goal of the K-median problem is to find a set of nodes (medians) with at most K elements in a complete graph G with edge lengths, such that the sum of the edge lengths from each node to the nearest median is minimized.

Let us assume an instance of the K-median problem in 2-D Euclidean space. Then, we can build an instance of the processing placement problem as follows. First, construct a network composed of all points given in the K-median problem. Second, set the traffic limits Γ to $\alpha(u) = \omega(u) = \delta(u,u) = 1$ for every node u while $\delta(u,v) = 0$ for $u \neq v$. With these limits, we can only allow sessions whose source and sink are the same with maximum traffic 1. Then, we also set both the cost $cost(u,v)$ and the distance $dist(u,v)$ of each edge (u,v) to be the distance $d(u,v)$ in 2-D Euclidean space. Then, the solution to this instance of the processing placement problem gives a set of K nodes, say R, that minimizes the dimensioning cost of

$$\sum_{(u,v)\in E} cost(u,v)cap(u,v)$$

Now, we prove that R is also the solution to the K-median problem.

First, because each session is of a form (x,x,b), and the triangle inequality holds for the entire network, a session (x,x,b) is configured always on the shortest path $\{(x,r),(r,x)\}$, which is also the least cost path given any $r \in R$ if x is not one of the processing location. (No links are used if the end point x is one of the processing nodes.) Note also that for any such link (x,r) or (r,x), the session form (x,x,b) is the only one that uses the links.

Therefore, the minimum capacity required at (x,r) or (r,x) is

$$cap(x,r) = cap(r,x) = min\{\alpha(x), \omega(x), \delta(x,x)\} = 1$$

From this,

$$\begin{aligned} & \sum_{(u,v)\in E} cap(u,v)cost(u,v) \\ = & \sum_{x\in V} cap(x,r) \min_{r\in R}(dist(x,r) + dist(x,r)) \\ = & \sum_{x\in V} \min_{r\in R}(d(x,r) + d(r,x)) \\ = & \; 2\sum_{x\in V} \min_{r\in R} d(x,r) \end{aligned}$$

This proves that R also minimizes the sum of the distance $\sum_{x\in V} \min_{r\in R} d(x,r)$, and thus is a solution to the K-median problem.

Therefore, the processing placement problem includes the K-median problem, and therefore is $\mathcal{NP}$-hard [12]. Moreover, the K-median problem is not approximable within any ϵ bound (not in $\mathcal{APX}$), however is approximable if the bound K can be relaxed. Lin and Vitter [15] presented an algorithm that gives a solution with the total distance within $2(1+\frac{1}{\epsilon})$ of the optimum if the size of the set R can be at most $(1+\epsilon)K$ in Euclidean space.

Unfortunately, the approximation algorithm does not provide a solution to the processing placement problem when we have arbitrary traffic constraints. Therefore, we need to

devise new algorithms that perform well under the conditions that we expect to arise in practice.

4.2.1. Selecting K nodes. In this section, we suggest two heuristic strategies for selecting the K nodes. First, we define the set of sessions with non-zero maximum traffic as follows.

$$S = \{(x, y) | t_{max}(x, y) > 0\}$$

where $t_{max}(x, y) = \min(\alpha(x), \omega(y), \delta(x, y))$, the maximum amount of traffic from x to y.

Next, we define the sessions that can be served by each processing location. So, for each location u,

$$S(u) = \{(x, y) | p(x, y, u) \leq f(x, y, \epsilon) \text{ and } (x, y) \in S\}$$

where $p(x, y, u)$ is the shortest path from x to y that goes through u.

Then, in the first strategy, we define the weight of each node pair (x, y) and the weight of each potential processing node u as

$$w(x, y) = t_{max}(x, y) l(x, y)$$

$$w(u) = \sum_{(x,y) \in S(u)} w(x, y)$$

where $l(x, y)$ is the length of the shortest path $p(x, y)$.

Now, we select K nodes that have the largest weight. This task can be carried out with the following greedy method. We start with an empty set R and a set S containing sessions with non-zero traffic limits and repeat the following steps until S is empty or $|R| = K$.

First, select a node u that gives the largest weight $w(u)$ and include it into R. Next, remove all sessions in $S(u)$ from S, i.e. $S = S - S(u)$. Note that this may change the weights of nodes that have not yet been selected. If, when the algorithm terminates, S is empty then the set R satisfies the bound on path lengths. Otherwise, it does not.

Finding such K nodes can also be formulated as an integer linear program, for which we can generalize a heuristic method introduced in [15] to obtain an approximated solution. Below, we give the linear program and leave the details of the approximation method for future work.

First, each constant variable $f_{i,j,h}$ for $u_i, u_j, u_h \in V$ is defined such that $f_{i,j,h} = 1$ if the node u_h can be designated as a processing node for sessions between the pair (u_i, u_j), i.e. $(u_i, u_j) \in S(u_h)$, and zero otherwise. Then, finding K nodes with the largest weight is equivalent to the following integer linear program that maximizes

$$\sum_{u_h \in V} \sum_{u_i, u_j \in V} f_{i,j,h} \times x_{i,j,h} \times y_h \times w(u_i, u_j)$$

subject to

$$\sum_{u_h \in V} f_{i,j,h} \times x_{i,j,h} \leq 1, \qquad u_i, u_j \in V$$
$$\sum_{u_h \in V} y_h \leq K$$
$$x_{i,j,h} \leq y_h$$
$$x_{i,j,h}, y_h \in \{0,1\}, \quad u_i, u_j, u_h \in V$$

where $y_h = 1$ if and only if the node u_h is chosen as a processing node, and $x_{i,j,h} = 1$ if and only if $y_h = 1$, $(u_i, u_j) \in S(u_h)$, and the pair (u_i, u_j) is assigned with the processing node u_h.

Overall, this heuristic attempts to place the processing nodes in favor of the end point pairs that are likely to produce more traffic and to require more resources. This is reflected in the weight of each potential processing node, which is the sum of the traffic contributions from the end point pairs that can be served by the node. Thus, this method is expected to to offer better (shorter) paths for end point pairs with large traffic contributions, and thus reduce the overall resource requirements.

Meanwhile, as an alternative strategy, we can define the weight $w(u)$, as the minimum dimensioning cost only considering end points in $S(u)$ and the processing node u, i.e. $w(u) = D(G, \{u\}, \Gamma)$. Similarly, we define the weight of a set of processing nodes R as the associated dimensioning cost, i.e. $w(R) = D(G, R, \Gamma)$ for end points in $\bigcup_{u \in R} S(u)$. With the new weights, we attempt to specify the portion of the optimal dimensioning cost contributed by each processing node, and select k nodes with the largest portion. Hopefully, we may identify the processing nodes that serve a majority of session traffic while approximating the optimal solution. Both the greedy and the linear programming method can be applied to this case.

We have proposed two strategies for heuristically selecting processing locations. Once we determine the k processing locations, we can dimension resources with the max flow formulation. In order to evaluate the strategies, we define the quality of a solution R as the dimensioning cost $D(G, R, \Gamma)$ relative to the optimal cost D_{opt}, i.e. $\frac{D(G,R,\Gamma)}{D_{opt}}$. However, without knowing the optimal solution and its cost, we can use the lower bound D_{lo} as an alternative way to measure the quality because

$$\frac{D(G, R, \Gamma)}{D_{lo}} \geq \frac{D(G, R, \Gamma)}{D_{opt}}$$

where the quality of the given solution is bounded by the left hand side.

The performance of the particular strategies for selecting K nodes and possibly other strategies need be studied further. We will measure and compare the methods in terms of the quality of the dimensioning cost and the performance of session configurations given the resources. As another variation, we may also try to tune the value of K as suggested in [15],

5. Concluding remarks

Programmable networks open up a broad range of ways to develop and operate applications by allowing customized processing in network routers. In this proposal, we have addressed two key issues that arise in operating and provisioning such networks.

First, we presented the problem of configuring application sessions that require intermediate processing on routers, and provided an optimal solution for a major class of sessions. We also discussed the issues related with configuring sessions that reserve resources, presented efficient algorithms and demonstrated experimentally that they can work well in practice.

Then, we considered how to provision a programmable network to satisfy anticipated resource demands. Particularly in the resource dimensioning problem, we generalized a formulation for dimensioning conventional networks so that it can be applied to programmable networks that support various types of processing. We then discussed the problem of placing processing resources where we illustrated the effect of different placements and potential methods for selecting optimal locations. By identifying problems in programmable networks and providing methods for resolving them, we hope to discover practical ways to realize programmable networks and benefit from them.

References

[1] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The switchware active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3), 1998.

[2] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, Apr. 1999.

[3] Y. Chae, S. Merugu, E. Zegura, and S. Bhattacharjee. Exposing the network: Support for topology sensitive applications. *Proceedings of IEEE OpenArch 2000*, March 2000.

[4] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Resource customizable management for value-added customizable network service. *IEEE Network*, 15(1), January 2001.

[5] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, , and M. Li. Approximation algorithms for directed steiner problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 192–200, January 1998.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1990.

[7] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner. A scalable, high performance active network node. *IEEE Network*, January/February 1999.

[8] A. J. Fingerhut. Approximation algorithms for configuring nonblocking communication networks. In *Washington University Computer Science Department doctoral dissertation*, May 1994.

[9] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness, [SP5].* W.H. Freeman and Company, 1979.

[10] R. Haas, P. Droz, and B. Stiller. Distributed service deployment over programmable networks. *International Workshop on Distributed Systems Operations and Management*, October 2001.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A packet language for active networks. *Proceedings of the International Conference on Functional Programming*, 1998.

[12] D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems.* PWS Publishing, 1996.

[13] R. Keller, J. Ramamirtham, T. Wolf, and B. Plattner. Active pipes: Program composition for programmable networks. *Proceedings of IEEE MILCOM 2001*, October 2001.

[14] A. Kershenbaum. *Telecommunications network Design Algorithms.* McGraw-Hill Book Company, 1993.

[15] J.-H. Lin and J. S. Vitter. ϵ-approximations with minimum packing constraint violation. In *Proceedings of 24th Annual ACM Symposium on Theory of Computing*, pages 771–782, 1992.

[16] H. Ma, I. Singh, and J. Turner. Constraint based design of atm networks, an experimental study. *Washington University Computer Science Department Technical Report WUCS-97-15*, 1997.

[17] R. Miller and Q. F. Stout. Algorithmic techniques for networks of processors. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 46. CRC Press, 1999.

[18] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. *Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[19] S. Ramanathan. Multicast tree generation in networks with asymmetric links. *IEEE/ ACM Transactions on Networking*, 4(4):558–568, 1996.

[20] G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. In *Symposium on Discrete Algorithms*, pages 770–779, 2000.

[21] M. Schwartz. *Computer-communication network design and analysis.* Prentice-Hall, 1977.

[22] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[23] P. Tullmann, M. Hibler, and J. Lepreau. Janos: A java-oriented os for active networks. *IEEE Journal on Selected Areas of Communication*, 19(3), Mar. 2001.

[24] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. *Proceddings of IEEE OPENARCH*, 1998.

[25] Y. Yemini, S. da Silva, D. Florissi, and H. Huang. The network flow language: A mark-based approach to active networks. *Technical Report, Columbia University Computer Science Department*, July 1999.